AUTOMATED INSTRUCTION-SET EXTENSION

Technical Field

[0001] The present invention is concerned with data processing techniques and, more particularly, with automated selection of application-specific extensions of an instruction set for an application processor.

Background of the Invention

[0002] For system-on-chip processors, progress has been made towards synthesis of application-specific-instruction-set processors (ASIP), including the generation of complete instruction sets for specific applications. Typically, the goal is the design of an instruction set for which a performance metric is optimized, e.g. minimization of run time, program memory requirements, or execution unit count. Application-specific instructions may be termed complex computer operations, including vector operations, fused operations, specialized operations and the like.

[0003] Recent attention has been given also to extending generic processors with units that are more broadly applicable, e.g. in a specified domain of applications. A typical goal of such processor extensions is optimization in the application domain without incurring the area and cost of top-of-the-line superscalar or multithreaded processors. An important motivation towards specialization of existing processors versus the design of complete ASIP's is to avoid the complexity of a complete processor and toolset development. Instead, an available and proven processor design and its extensible toolset can be leveraged, with design efforts focused on a special datagraph.

[0004] One method for generating an instruction-set extension from a description of an application is described in published U.S. Patent Application No. US 2003/0074654 by Goodwin et al., entitled "Automatic Instruction Set Architecture". Described there is a basic fusion algorithm.

Summary of the Invention

[0005] An instruction-set extension for an application can include clusters of

dataflow operations, e.g. for system-on-chip processors. We have recognized that, in choosing clusters, micro-architectural constraints can be taken into account, and a legality property can be enforced. Multiple-output instructions can be selected, as register-file write-port constraints are met. In a dataflow graph of the application, generic disconnected subgraphs can be identified for implementation as instruction-set extensions.

Brief Description of the Drawing

[0006] Fig. 1 is a dataflow graph of the basic block most frequently executed in a typical embedded processor benchmark.

[0007] Fig. 2 is a representation of a subgraph of the dataflow graph that does not satisfy a convexity requirement.

[0008] Fig. 3 is a representation of a search tree corresponding to the subgraph shown in Fig. 2.

[0009] Fig. 4 is a listing of pseudo-code for an identification method.

[0010] Fig. 5 is a representation of an execution trace for the graph of Fig. 2.

[0011] Fig. 6 is a plot of the number of cuts considered.

[0012] Fig. 7 is a representation of a search tree for two cuts.

[0013] Fig. 8 is a representation of cuts in optimal selection of three cuts in three basic blocks.

Detailed Description

[0014] The dataflow graph of Fig. 1 is a simple yet realistic example used here for motivational illustration. In the graph, SEL represents a selector node resulting from applying an if-conversion pass to the code underlying the graph.

[0015] On inspection of the graph it is apparent that identification based on recurrence of clusters is unlikely to find candidates of more than 3 to 4 operations. Apparent further are recurring clusters, e.g. cluster M0 having multiple inputs which may be prohibitive. Choosing larger, though non-recurrent clusters might ultimately reduce

the number of inputs and/or outputs as is the case for subgraph M1 which satisfies even the most stringent constraints of two operands and one result. Inspection of original code suggests that this subgraph represents an approximate 16 by 3-bit multiplication, likely to be chosen by a designer even under severe area constraints. Availability of a further input would also include the subsequent accumulation and saturation operations per subgraph M2. If additional inputs and outputs are available, it would be desirable to implement both M2 and M3 as part of the same instruction, thus exploiting the parallelism of the two disconnected graphs.

[0016] For an abstract statement of aims, the following notation is used in the following:

[0017] G(V, E) denotes a directed acyclic graph (DAG) representing the dataflow of a basic block, where the nodes V represent primitive operations and the edges E represent data dependencies. Each graph G is associated with a graph G+ (V union V+, E union E+) containing additional nodes V+ and edges E+. The additional nodes V+ represent input and output variables of the basic block. The additional edges E+ connect nodes V+ to V and nodes V to V+.

[0018] A cut is a subgraph of G. There are 2|V| possible cuts, where |V| is the number of nodes in G. A function M(S) is chosen as a measure of merit for a cut S. It serves as an objective function to be optimized, e.g. an estimate of speedup achievable by implementing S as a special instruction.

[0019] IN(S) denotes the number of predecessor nodes of those edges which enter the cut S from the rest of the graph G+, i.e. the number of inputs to the operations in S. Correspondingly, OUT(S) is the number of predecessor nodes in S from which edges exit the cut S. They represent the number of outputs from S to other operations, either in G or another basic block. A cut S is called convex if there exists no path from a node *u* in S to another node *v* in S involving a node *w* not in S. For contrast, Fig. 2 shows an example of a non-convex cut.

[0020] With each basic block considered independently, an identification problem can be formally stated as follows:

[0021] Problem 1. Given a graph G+, find the cut S which maximizes M(S) under the following constraints: 1. IN(S) $\leq$ Nin, 2. OUT(S) $\leq$ Nout, and 3. S is convex.

[0022] Chosen values Nin and Nout indicate the register-file read and write ports, respectively, which can be used by the special instruction. The convexity constraint is a legality check on the cut S and serves to ensure that a feasible solution exists. As illustrated by Fig. 2, if all inputs of an instruction are to be available at issue time, and all results are produced at the end of instruction execution, there is no possible schedule which can respect the dependencies of this graph once S is collapsed into a single instruction.

[0023] Several special instructions from all basic blocks will be allowed, with Ninst denoting the maximum number of cuts which together give the maximum advantage. For final selection of Ninst cuts, a heuristic approach can involve repeatedly solving Problem 1 on all basic blocks and by selecting the Ninst best ones. A formal statement is as follows:

[0024] Problem 2. Given the graphs Gi+ of all basic blocks, find up to Ninst cuts Sj which maximize $\sum j$ M(Sj) under the constraints 1 to 3 of Problem 1 for each cut Sj.

[0025] A novel method for solving Problems 1 and 2 can be described in terms of the following three steps: (1) find the optimal single cut in a single basic block, (2) find an optimal set of non-overlapping cuts in several basic blocks, and (3) find a near-optimal set of non-overlapping cuts in several basic blocks.

[0026] Exhaustively enumerating all possible cuts within a basic block may not be computationally feasible in practice, but a method described in the following can be used to explore the complete search space while effectively detecting and pruning infeasible regions during the search. The method starts with a topological sort on G. Nodes of G are ordered such that if G contains an edge (u, v) then u appears after v in the ordering. For illustration, Fig. 2 shows a topologically sorted graph. The method uses a recursive search function based on this ordering to explore a search tree.

[0027] Fig. 3 shows the search tree for the example of Fig. 2, with some of the tree nodes labeled with their cut values. The search tree is a binary tree of nodes

representing possible cuts. It is built from a root representing the empty cut, and each pair of 1- and 0-branches at level *i* represents addition, yes or no, of the node of G having topological order *i*, to the cut represented by the parent node.

[0028] Nodes of the search tree immediately following a 0-branch represent the same cut as their parent node, and can be ignored in the search. The search proceeds as a preorder traversal of the search tree. In some cases there is no need to branch towards lower levels, so that the search tree is pruned.

[0029] For instance, when the output port constraint has already been violated by a cut defined by a certain tree node, then adding nodes that appear later in the topological ordering cannot reduce the number of outputs of the cut. Similarly, if the convexity constraint is violated at a certain tree node, there is no way of regaining feasibility by inserting nodes of G that appear later in the topological ordering. For example, with reference to Fig. 2, after inclusion of node 3 the only ways to regain convexity are to either include node 2 or remove from the cut node 0 or node 3. Because of the topological ordering, both actions are precluded in a search step subsequent to insertion of node 3. As a consequence, when the output-port or the convexity constraint is violated when reaching a certain search tree node, then the subtree rooted at that node can be eliminated from the search space.

[0030] Violation of the input constraint can be exploited also for pruning. For Example, for the search tree node where node 0 has been included and nodes 1 and 2 have been considered and excluded, the two inputs of node 0 can no longer be eliminated. Thus, an input constraint of 1 would allow pruning of the subtree.

[0031] Fig. 4 shows the above method as represented in pseudo-C notation. The search tree is implemented implicitly by use of the recursive *search(.)* function. The parameter *current_choice* defines the direction of the branch, and the parameter *current_index* defines the index of the graph node and the level of the tree on which the branch is taken. When the output-port check or the convexity check fails, or when a leaf is reached during the search, the method backtracks. The best solution is updated only if all constraints are satisfied by the current cut.

[0032] Fig. 5 shows application of the method to the graph of Fig. 2, with Nout = 1. Only 5 cuts pass both output-port check and convexity check, while 6 cuts are found to violate either output-port constraint or convexity constraint, resulting in elimination of 4 more cuts. Therefore, among 16 possible cuts only 11 are considered.

[0033] The graph nodes contain O(1) entries in their adjacency lists on average, as the number of inputs for a graph node is limited in every practical case. Combined with a single node insertion per method step, the *input_port_check*, *output_port_check*, *convexity_check* and *calculate_speedup* functions can be implemented in O(1) time using appropriate data structures. Thus, the over-all complexity of the method is O(2|V|). Although still exponential, the method in practice reduces the search significantly.

[0034] Fig. 6 shows run-time performance of the method using an output-port constraint of two on some basic blocks extracted from several benchmarks. An exponential tendency is perceptible, but the actual performance is within polynomial bounds in all practical cases considered. Constraint-based subtree elimination plays an important role in the method's performance. The tighter the constraints are, the faster the method.

[0035] Problem 3. Given a graph G+, find the cut which maximizes M(S) under the constraints 1-3 of Problem 1, and under the further constraint 4. S consists of a single connected graph.

[0036] Problem 3 can be solved optimally by the method described above with reference to Fig. 3, with an additional pruning potentiality as follows. Every time a maximal connected subgraph is found in a cut under consideration, i.e. a subgraph that cannot grow further without becoming disconnected, the subtree rooted at that node can be pruned.

[0037] On account of additional pruning, an optimal solution to Problem 3 can be found considerably faster as compared with Problem 1. On the other hand, if for a certain graph G and certain input/output constraints the optimal solution to Problem 1 resulted in a disconnected graph, then solving Problem 3 on the same graph and under the same constraints would not yield the same solution. An efficient compromise can be stated as

follows:

[0038] Problem 4. Given a graph G+, find the cut which maximizes M(S) under the constraints 1-3 of Problem 1, and under the further constraint 4'. S does not include any maximal connected graphs S_I with $M(S\_I) < \alpha \cdot M(S\_Prob\_3)$ where M(S_Prob_3) is the merit of the optimal solution of Problem 3, being a single connected graph.

[0039] The parametric factor α has a value between 0 and 1, chosen experimentally for example. With $\alpha = 0$ in Problem 4, Problem 1 can be viewed as a special case of Problem 4.

[0040] Problem 4 can be solved optimally the method described above with reference to Fig. 3, with an additional pruning potentiality as follows. Every time a maximal connected subgraph is found in a cut under consideration, i.e. a subgraph that cannot grow further without becoming disconnected, the subtree rooted at that node can be pruned provided its gain is less than $\alpha \cdot M(S\_Prob\_3)$. For this purpose, the value of M(S_Prob_3) can be found by prior solution of Problem 3.

[0041] In trials with $\alpha = 0.7$, the method was applied successfully to Problem 4 on graphs of up to 400 nodes, while solving Problem 1 for graphs beyond 100 nodes became increasingly impracticable. Where comparison was feasible, the difference in gain of the two solutions was insignificant.

[0042] A method as described above is adaptable for determining multiple cuts from a single graph. If M is the number of cuts to be identified within a basic block, it is sufficient to establish a similar search tree where every node makes M + 1 branches instead of 2. Fig. 7 shows a fragment of a tree for M = 2. Nodes of the search tree now represent M cuts. An n-branch at level *i* results in inclusion of the graph node with index *i* in the *n*-th cut.

[0043] For optimal selection, the method begins by applying the single-cut identification method on each basic block (M = 1). The first cut is chosen from that basic block which offers the greatest speed-up improvement. Then, at each iteration, the method increments the value of M for the basic block which was chosen by the previous iteration, performs multi-cut identification on this basic block with the new value of M,

and determines the improvement. Again the new cut is chosen from the basic block that gives the greatest speed-up improvement. The iteration continues until Ninstr cuts have been chosen. The method yields an optimal solution upon applying the multiple-cut identification method at most Ninstr + Nbb – 1 times where Nbb denotes the number of blocks. Fig. 8 illustrates the method's use in a simple case with three blocks.

[0044] Repeated execution of the multiple-cut identification method on large blocks can result in impracticable computational complexity. Remedially, a heuristic approach can be used, with iterative applications of the single-cut identification method to the same basic block. Previously identified cuts are merged into single graph nodes, and are excluded from subsequent identification steps.

[0045] A practical setting for benefiting from the described techniques can involve an initial high-level code for an application, compilation of the high-level code into intermediate-level code, and, from the intermediate-level code, automated generation of instruction-set extensions under one or several constraints as described above. The instruction-set extensions can be specified in VHDL, for example (VHDL: VHSIC (Very High Speed Integrated Circuit) Hardware Description Language). A standard Synopsis design tool, or a tool chain such as from LisaTek (Coware) or from Tensilica can synthesize the specification onto hardware.

[0046] Techniques as described above, optimal as well as iterative, were implemented within the MachSUIF framework of M. D. Smith et al., An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization, Harvard University Press, Cambridge, Mass., 2000. The implementations were tested on a subset of the MediaBench suite benchmarks of C. Lee et al., A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, Proceedings of the 30th Annual International Symposium on Microarchitecture, pp. 330-335, Research Triangle Park, N.C., December 1997. Application C-code is translated to MachSUIF intermediate representation and pre-processed with a standard if-conversion pass. The results were compared with those of a greedy linear-complexity method that can detect n-input, m-output graphs, where *n* and *m* are specified parameters. The results were compared

further with a linear complexity method that identifies single-input and unbounded-input graphs.

[0047] In the tests, the difference between optimal and iterative methods was found to insignificant, with both outperforming the prior-art methods. For low input/output constraints, performed comparably, but in the case of higher, still very reasonable constraints, the iterative method excelled. As compared with the prior art, for the present methods there is a significant potential performance advantage for multiple-output and generally disconnected graphs.